Online Trichromatic Pickup and Delivery Scheduling in Spatial Crowdsourcing

Zheng, Bolong; Huang, Chenze; Jensen, Christian S.; Chen, Lu; Hung, Nguyen Quoc Viet; Liu, Guanfeng ; Li, Guohui; Zheng, Kai

Published in:
2020 IEEE 36th International Conference on Data Engineering

DOI (link to publication from Publisher):
10.1109/ICDE48307.2020.00089

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Online Trichromatic Pickup and Delivery Scheduling in Spatial Crowdsourcing

Bolong Zheng[1], Chenze Huang[2], Christian S. Jensen[3], Lu Chen[3], Nguyen Quoc Viet Hung[4], Guanfeng Liu[5], Guohui Li[1,*], Kai Zheng[6]
[1]Huazhong University of Science and Technology, [2]Sun Yat-sen University, [3]Aalborg University
[4]Griffith University, [5]Macquarie University, [6]University of Electronic Science and Technology of China
{bolongzheng, guohuili}@hust.edu.cn, huangchz@mail2.sysu.edu.cn, {csj, luchen}@cs.aau.dk,
quocviethung.nguyen@griffith.edu.au, guanfeng.liu@mq.edu.au, zhengkai@uestc.edu.cn

***Abstract*—In Pickup-and-Delivery problems (PDP), mobile workers are employed to pick up and deliver items with the goal of reducing travel and fuel consumption. Unlike most existing efforts that focus on finding a schedule that enables the delivery of as many items as possible at the lowest cost, we consider trichromatic (worker-item-task) utility that encompasses worker reliability, item quality, and task profitability. Moreover, we allow customers to specify keywords for desired items when they submit tasks, which may result in multiple pickup options, thus further increasing the difficulty of the problem. Specifically, we formulate the problem of Online Trichromatic Pickup and Delivery Scheduling (OTPD) that aims to find optimal delivery schedules with highest overall utility. In order to quickly respond to submitted tasks, we propose a greedy solution that finds the schedule with the highest utility-cost ratio. Next, we introduce a skyline kinetic tree-based solution that materializes intermediate results to improve the result quality. Finally, we propose a density-based grouping solution that partitions streaming tasks and efficiently assigns them to the workers with high overall utility. Extensive experiments with real and synthetic data offer evidence that the proposed solutions excel over baselines with respect to both effectiveness and efficiency.**

***Index Terms*—Spatial Crowdsourcing, pickup and delivery, scheduling, real-time, query optimization**

I. Introduction

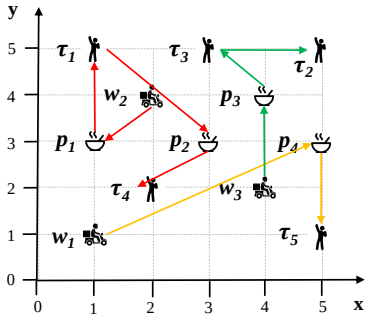
With the prevalence of mobile Internet access and the growth of the sharing economy, we have witnessed the advent of different spatial crowdsourcing (SC) platforms. In many real-world SC applications, online crowd workers are assigned tasks through their smartphones, two examples being real-time transportation tasks as offered by, e.g., Uber [5] and Didi [1], and supermarket product placement checking tasks as offered by, e.g., Gigwalk [3]. Further, online food ordering and delivery services (e.g., Meituan [4], Ele [2], and Ubereats [6]) are gaining popularity. Using these services, customers can search for a favorite restaurant, usually filtered by type of cuisine, and can choose from available items. Their order is then delivered by a worker assigned by the service. According to one report [8], as of September 2016, online delivery accounted for about 3 percent of 61 billion U.S. restaurant transactions.

Many SC problems that require crowd workers to retrieve and deliver sets of items are instances of the Pickup-and-Delivery Problem (PDP) in which crowd workers service spatially located requests. Given a set of item pickup and delivery locations along with a set of crowd workers, the goal is to find a schedule for the crowd workers that minimizes the energy consumed or the delivery time, given constraints such as time windows and capacities [13]. The PDP is NP-hard and has been studied extensively. An existing study [7] offers a detailed review of general issues as well as solution strategies of dynamic pickup and delivery problems where items have to be collected and delivered in real-time.

Going beyond the classical application scenario, the emergence of real-world applications (e.g., Meituan) brings new requirements and challenges to existing PDP techniques in SC platforms. For instance, the pickup location of a desired item may not be a single location. A customer may simply order "Kung Pao Chicken", which offers the worker the possibility of selecting any among multiple nearby Chinese restaurants as the pickup location. Moreover, a customer's level of satisfaction is mainly affected by their restaurant preferences (e.g., the quality, price and rating), the worker (e.g., rating), and the waiting time. For example, customers may prefer to order food from nearby restaurants with higher ratings and have it delivered quickly by a reliable worker. In addition, workers may receive extra rewards for completing urgent tasks in a timely manner. Therefore, SC platforms can stimulate workers by assigning higher rewards to tasks. Consequently, a clear trend for SC platforms is to promote their businesses by attempting to maximize the overall satisfaction of both customers and workers.

To this end, we study a novel generalization of the pickup and delivery problem that takes both utility and keywords into consideration, namely the *online trichromatic pickup and delivery scheduling* (OTPD) problem. In real-world settings, a set of pickup Point-of-Interests (PoIs) are available from which customers can order their items online. Each PoI is associated with a set of keywords that describes the available items, and a score which can either be the rating that captures the quality of its services/items, or simply the price. The orders or tasks submitted by customers are received over time. Each consists of a destination, a set of keywords describing the desired items, a task radius that denotes the region of the items that are considered, an expiration time before which the task must be

*Corresponding author: Guohui Li



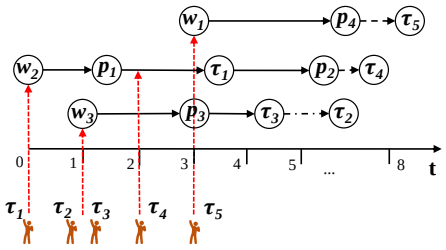
(a) Original locations

Task	τ.t	τ.e	τ.ϕ	τ.r
τ_1	0	6	{c}	0.4
τ_2	1	7	{a}	0.5
τ_3	1	7	{e}	0.6
τ_4	2	10	{b}	0.8
τ_5	3	10	{d}	0.6

PoI	$p.\Phi$	$p.r$
p_1	{c,d}	0.7
p_2	{a,b,c}	0.5
p_3	{a,c,e}	0.6
p_4	{b,d}	0.8

Worker	w.r
w_1	0.3
w_2	0.5
w_3	0.7

(b) Task, worker, and pickup PoI information



(c) Task schedule

Fig. 1. Running Example

completed, and an extra reward that the customer would like to offer. A set of workers are available to conduct delivery tasks assigned by the SC platform or server, and each completed task is rated by the customer involved. Each worker has a radius that denotes the region of delivery locations serviced by the worker. For each worker-item-task match, a trichromatic matching utility is proposed to comprehensively evaluate the satisfaction with the worker's reliability, the item's quality, and the task's profitability. When new tasks are submitted, the OTPD problem is to immediately assign the tasks to workers with the goal of maximizing the overall utility, subject to given spatial, temporal, keyword, and capacity constraints.

Example 1. *Fig.1 shows a running example. Fig.1(a) indicates the locations of each worker, pickup PoI, and customer. Fig.1(b) gives information on the tasks* $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$*, the pickup PoIs* $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$*, and the workers* $\mathcal{W} = \{w_1, w_2, w_3\}$*. Fig.1(c) shows the time-ordered task schedule. The capacity of each worker is 2. The unified task and worker radii are set to 4. As will be explained in Section II-B, we use* $\alpha = 0.3$ *and* $\beta = 0.3$ *to compute the utility for a worker-item-task match.*

At time 0, τ_1 *is submitted with expiration time* $\tau_1.e = 6$*, the desired item is described by* $\tau_1.\phi = \{c\}$*, and the extra reward* $\tau_1.r$ *is 0.4. As the task and worker radii are 4,* p_1 *and* p_2 *are candidate pickup PoIs, and* w_1 *and* w_2 *are candidate workers. The OTPD assigns* τ_1 *to* w_2 *that picks up item* $\{c\}$ *at* p_1 *with utility* $\mu(w_2, p_1, \tau_1) = 0.52$*. At time 1, both* τ_2 *and* τ_3 *are submitted. Task* τ_2 *is assigned to* w_3 *that picks up* $\{a\}$ *at* p_3 *with utility* $\mu(w_3, p_3, \tau_2) = 0.59$*, and* τ_3 *is also assigned to* w_3 *that picks up* $\{e\}$ *at* p_3 *with utility* $\mu(w_3, p_3, \tau_3) = 0.63$*. At time 2,* τ_4 *is submitted and assigned to* w_2 *that picks up* $\{b\}$ *at* p_2 *with the utility* $\mu(w_2, p_2, \tau_4) = 0.62$*. Note that* w_2 *has already picked up* $\{c\}$ *from* p_1 *at the time* τ_4 *is accepted. At time 3,* τ_5 *is submitted and assigned to* w_1 *that picks up* $\{d\}$ *at* p_4 *with utility* $\mu(w_1, p_4, \tau_5) = 0.57$*.*

From Fig.1(c), we can see that the delivery schedule for w_1 *is* $w_1 \rightarrow p_4 \rightarrow \tau_5$*, for* w_2 *is* $w_2 \rightarrow p_1 \rightarrow \tau_1 \rightarrow p_2 \rightarrow \tau_4$*, and for* w_3 *is* $w_3 \rightarrow p_3 \rightarrow \tau_3 \rightarrow \tau_2$*. Therefore, the overall utility of the schedule is* 2.93*.*

In particular, as the OTPD problem is NP-hard and difficult to address. Therefore, we first propose a greedy algorithm that finds a schedule with the highest utility-cost ratio for each upcoming task while taking both the utility and incremental travel cost into account such that more tasks are likely to be served. Moreover, we develop a novel index structure called skyline kinetic tree (SK-tree), which stores not only a selected schedule, but also all its skyline schedules for each worker to execute. A key technical novelty is that the SK-tree materializes these intermediate results and improves the overall utility. Finally, in order to process tasks in a batch, we exploit a density-based grouping method that partitions the tasks from a short timespan into different groups and forms the matches with small incremental travel cost within a group, thus avoiding the overhead of enumerating all combinations.

In brief, the key contributions are summarized as follows:

- We formalize the Online Trichromatic Pickup and Delivery Scheduling (OTPD) problem.
- We propose a greedy assignment algorithm that finds the schedule with the highest utility-cost ratio. In order to improve the result quality, we introduce a skyline kinetic tree-based algorithm to materialize and reuse intermediate results. Finally, we propose a density-based grouping algorithm to handle the tasks in a batch.
- We conduct extensive experiments with real and synthetic datasets, providing evidence that the proposed solutions excel over baselines with respect to both effectiveness and efficiency.

The rest of the paper is organized as follows. We formulate the OTPD problem in Section II. Section III introduces the system framework. We develop the greedy assignment algorithm in Section IV and the skyline kinetic tree-based algorithm to materialize intermediate results in Section V. We introduce the density based grouping algorithm in Section VI. Section VII reports on the experimental study. We review related work in Section VIII and conclude in Section IX.

II. Preliminaries

We proceed to define the OTPD problem. Frequently used notation is summarized in Table I.

TABLE I
SUMMARY OF NOTATION

Notation	Definition
$\tau = (l, t, e, \phi, \rho, r)$	A delivery task τ
$w = (l, t, c, \rho, r)$	A worker w
$p = (l, \Phi, r)$	A pickup PoI p
$\mu(w, p, \tau)$	The utility of of a match (w, p, τ)
$\mathcal{A}(w, P, T)$	A worker w with tasks T and PoIs P
$\mathcal{DS}(w)$	A pickup and delivery schedule for w
$\mathsf{dist}(\mathcal{DS}(w))$	The travel distance of $\mathcal{DS}(w)$
$\mathsf{cost}(\mathcal{DS}(w))$	The travel cost of $\mathcal{DS}(w)$
$\mathcal{U}(\mathcal{DS}(w))$	The overall utility of $\mathcal{DS}(w)$

A. Settings

Definition 1 (Pickup and Delivery Task). *A pickup and delivery task $\tau = (l, t, e, \phi, \rho, r)$ is submitted by a customer at time $\tau.t$ to pickup an item $\tau.\phi$ and deliver it to a destination $\tau.l$ before an expiration time $\tau.e$. In a task, $\tau.\phi$ is a set of keywords that indicate the type of the item to be delivered, $\tau.\rho$ is the radius of a region centered at $\tau.l$ that denotes the region of the items that are considered, and $\tau.r \in [0, 1]$ is an extra reward offered for successfully completing task τ.*

Definition 2 (Worker). *A worker $w = (l, t, c, \rho, r)$ is an entity who is able to deliver items, where $w.l$ is the location of w at the current time $w.t$. Worker w has task capacity $w.c$ that denotes the maximum number of active tasks the worker can handle at the same time. Furthermore, $w.\rho$ is the radius of the circular region centered at $w.l$ that denotes the region of delivery locations serviced by the worker, and $w.r \in [0, 1]$ is the rating of the worker given by the customers.*

An active worker continuously sends inquiries to the server, including the up-to-date location $w.l$ at $w.t$. A worker only accepts the task τ whose destination $\tau.l$ is no further than $w.\rho$ from $w.l$. Using this information, the server assigns tasks and makes delivery plans for the workers. We assume unified settings for task radius $\tau.\rho$ and worker radius $w.\rho$ for all delivery tasks and workers. However, the proposed algorithms can be generalized easily to task-specific and worker-specific constraints.

Definition 3 (Pickup Point-of-Interest). *A pickup Point-of-Interest (PoI) $p = (l, \Phi, r)$ is a place that provides items for customers, where $p.l$ is the location of the PoI, $p.\Phi$ is a set of keywords describing the item types provided by p, and $p.r \in [0, 1]$ is a score of p, which can be either rating or price.*

B. Trichromatic Matching Utility

Existing delivery services often offer the customers only one choice that aims to minimize the delivery time. However, the customer experience is usually affected by many factors, such as the quality or price of the item and the service quality of the worker involved in a delivery. For example, customers may prefer to order food from restaurants with higher ratings and have it delivered by a reliable worker. In addition, a higher extra reward may attract more workers to bid for a task.

Definition 4 (Trichromatic Match). *Given a worker w, a pickup PoI p, and a task τ, if w can pickup an item from p to successfully complete τ, the triple (w, p, τ) forms a valid trichromatic (worker-item-task) match. The trichromatic matching utility is defined as follows:*

$$\mu(w, p, \tau) = \alpha \times w.r + \beta \times p.r + (1 - \alpha - \beta) \times \tau.r, \quad (1)$$

where $\alpha, \beta \in [0, 1], \alpha + \beta \leq 1$, are balancing parameters that can be specified by customers or by the system. Intuitively, a higher α indicates that customers prefer a worker with a higher reliability. Likewise, a higher β indicates that customers want to order items from PoIs with high quality or low price. A higher $(1 - \alpha - \beta)$ means that customers prefer to pay more to attract workers such that the task is accepted. This may be especially important during peak hours.

Two existing studies [12], [22] also evaluate the matching quality with trichromatic utilities. One of these studies [22] argues that any function derived from profiles and spatial-temporal information related to worker-item-task is supported. Therefore, our definition can be considered as a specialization that uses a weighted average of the three factors (worker-item-task). The other study [12] focuses on ridesharing and also adopts the weighted average approach, but the three factors are computed differently from how we compute them. In particular, their vehicle-related utility is similar to our worker reliability; their rider-related utility captures the similarity between riders on the same vehicle, while we do not consider the connection between items of the same worker, but compute an independent score for each item; and their trajectory-related utility captures the detour of each rider due to the ridesharing. However, no matter how utility is computed, given a trichromatic worker-item-task match, the utility must be readily available to scheduling algorithm.

C. Task Schedule

1) Valid Schedule Constraints: Assume a worker w is assigned a set of m tasks T and a corresponding set of m pickup PoIs P. We denote this assignment as $\mathcal{A}(w, P, T) = \{(w, p_i, \tau_i) \mid p_i \in P, \tau_i \in T\}$, which contains m matches. A static pickup and delivery schedule of $\mathcal{A}(w, P, T)$ is a sequence of $3m$ events, denoted as $\mathcal{DS}(w) = \langle x_1, \ldots, x_{3m} \rangle$, where each $x_i \in \mathcal{DS}(w)$ is either an acceptance event (the **worker** accepts a task), a pickup event (the worker pickups an **item**), or a delivery event (the worker completes the **task**). For simplicity, we assume that the travel speeds of all workers are set to the same constant so that travel distance $\mathsf{dist}(\mathcal{DS}(w))$ and travel cost $\mathsf{cost}(\mathcal{DS}(w))$ can be used interchangeably. In addition, the travel distance can be either the Euclidean distance or the road-network distance. For simplicity, we use Euclidean distances, but road-network distance can be accommodated easily. The cost of $\mathcal{DS}(w)$ is computed as follows:

$$\mathsf{cost}(\mathcal{DS}(w)) = \sum_{i=2}^{3m} \mathsf{cost}(x_{i-1}.l, x_i.l), \quad (2)$$

where $x_i.l$ is the location of event x_i. The utility of $\mathcal{DS}(w)$ is computed as follows:

$$\mathcal{U}(\mathcal{DS}(w)) = \sum\nolimits_{p_i \in P, \tau_i \in T} \mu(w, p_i, \tau_i) \tag{3}$$

Valid Schedule. A static pickup and delivery schedule $\mathcal{DS}(w)$ is valid if it satisfies all the following constraints:

(i) **Task acceptance constraint**. For a new task τ, a trichromatic match (w, p, τ) is valid iff τ's destination is in the circular region with radius $w.\rho$ centered at $w.l$, according to Definition 2. If x_i is the acceptance event w.r.t. w and x_k is the delivery event w.r.t. τ, we have $\mathsf{dist}(x_i.l, x_k.l) \leq w.\rho$.

(ii) **Pickup selection constraint**. For a new task τ, a worker-item-task match (w, p, τ) is valid iff p is in the circular regions with radius $\tau.\rho$ centered at $\tau.l$, and p offers the item type specified by τ, according to Definition 1. If x_j is the pickup event w.r.t. p and x_k is the delivery event w.r.t. τ, we have $\mathsf{dist}(x_j.l, x_k.l) \leq \tau.\rho$, and $\tau.\phi \subseteq p.\Phi$.

(iii) **Travel order constraint**. For any task τ in $\mathcal{DS}(w)$, let x_j be the pickup event and x_k be the delivery event of τ. Then, we have $j < k$, i.e., the pickup must happen before the delivery.

(iv) **Task capacity constraint**. The number of active tasks (the item has already been picked up, but not yet delivered) of the worker w at any time does not exceed the capacity $w.c$.

(v) **Expiration time constraint**. The item of any task τ must be delivered to its destination $\tau.l$ before the time $\tau.e$.

2) Dynamic Pickup and Delivery Schedule: A static schedule $\mathcal{DS}(w)$ is a time-ordered event sequence for a worker w. However, an online pickup and delivery system needs to respond to each customer/task immediately such that when a task is submitted to the system, the task is quickly assigned to a worker. Therefore, we maintain a dynamic schedule $\mathcal{DS}(w) = \langle x_0, x_1, \ldots, x_n \rangle$, to capture the up-to-date status of each worker w. Assume that a new task τ is submitted to the system and worker w accepts τ. Then, x_0 is the acceptance event of w w.r.t. τ, and each x_i, $i \in [1, n]$, is either a pickup event or a delivery event since all previous acceptance events are obsolete. Specifically, we denote the pickup and delivery events of a task τ as $x_\tau^{\triangleright}$ and $x_\tau^{\triangleleft}$, respectively. In order to insert a new task into the current schedule, we provide an appropriate structure for $\mathcal{DS}(w)$. Unlike the related transfer event structure [12] that uses a segment-based representation, we use a point-based representation for $\mathcal{DS}(w)$ that we believe is more intuitive and easy to understand. Specifically, for each event x_i, we store the following information:

a) the location $x_i.l$;
b) the earliest arrival time $x_i.t^-$;
c) the latest leaving time $x_i.t^+$;
d) the task load $x_i.R$ after the completion of x_i.

The slack time for each event x_i is $t_i = x_i.t^+ - x_i.t^-$, and the worker w is allowed to spend some time between 0 and t_i to conduct the pickup or delivery. Fig. 2 shows the information

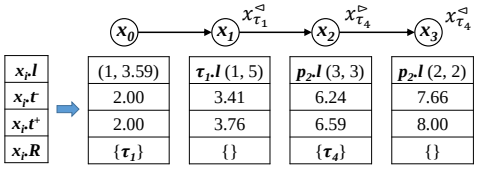


Fig. 2. Delivery Schedule $\mathcal{DS}(w_2)$

of $\mathcal{DS}(w_2)$ from the previous example where, at time 2, w_2 accepts τ_4.

If we assume that the worker does not waste any time on previous pickup and delivery events, i.e., no slack time is used, the earliest arrival time $x_i.t^-$ of each event x_i is computed as follows:

$$x_i.t^- = x_0.t^- + \sum\nolimits_{j=0}^{i-1} \mathsf{cost}(x_j.l, x_{j+1}.l) \tag{4}$$

Let $x_i.dl$ be the deadline of an event x_i. The deadline of a delivery event $x_\tau^{\triangleleft}$ w.r.t. a task τ is simply the expiration time, i.e., $x_\tau^{\triangleleft}.dl = \tau.e$. For a pickup event $x_\tau^{\triangleright}$ with a corresponding delivery event $x_\tau^{\triangleleft}$, the deadline of $x_\tau^{\triangleright}$ is computed as follows:

$$x_\tau^{\triangleright}.dl = x_\tau^{\triangleleft}.dl - \mathsf{cost}(x_\tau^{\triangleright}.l, x_\tau^{\triangleleft}.l) \tag{5}$$

If we assume that the worker uses up all the previous slack time, the latest leaving time $t_{x_i}^+$ of each event x_i is computed as follows:

$$x_i.t^+ = \min \left\{ \begin{array}{c} x_i.dl \\ x_{i+1}.t^+ - \mathsf{cost}(x_i.l, x_{i+1}.l), \end{array} \right. \tag{6}$$

which is used to guarantee that it is possible to complete all the events after x_i. Specifically, for x_0, we have $t_0 = x_0.t^- = x_0.t^+ = x_0.dl$; and for x_n, we have $x_n.t^+ = x_n.dl$. Note that if we insert new events into $\mathcal{DS}(w)$ or if worker w spends any slack time for any event x_i, the earliest arrival time and the latest leaving time of all the relevant events must be updated.

D. Problem Definition

Given a set of pickup PoIs $\mathcal{P}$, a set of pickup and delivery tasks $\mathcal{T}$ that arrive in real-time, and a set of crowd workers $\mathcal{W}$, the Online Trichromatic Pickup and Delivery Scheduling (OTPD) problem is to find an assignment $\mathcal{A}(w_i, P_i, T_i)$ and build a delivery schedule $\mathcal{DS}(w_i)$ for each w_i with small cost, where $w_i \in \mathcal{W}, T_i \subseteq \mathcal{T}, P_i \subseteq \mathcal{P}$, and $\forall i \neq j, T_i \cap T_j = \phi$, such that the overall utility, i.e.,

$$\sum\nolimits_{w_i \in \mathcal{W}} \mathcal{U}(\mathcal{DS}(w_i)) \tag{7}$$

is maximized, subject to all $\mathcal{DS}(w_i)$ being valid.

Ridesharing, which enables drivers to share empty seats in their vehicles with riders, can be considered as a special case of the OTPD problem, where there is only one pickup location w.r.t. a task. However, unlike the goal of maximizing the utility, ridesharing studies [16], [17], [20], [21], [23] mainly focus on matching drivers and riders on a real-time basis to minimize the overall travel cost. One study [11] considers both the travel cost and price of ridesharing and identifies the resulting skyline of options. While one study [22] considers the

similar problem of maximizing the utility, our problem differs in two main aspects: (1) we consider not only the matching, but also the scheduling for each worker; (2) we enable pickup options, i.e., an item has a set of possible pickup PoIs.

Theorem 1. *The OTPD problem is NP-hard.*

Proof. We consider a special instance of the OTPD problem where each worker w's schedule is empty and the task capacity $w.c$ is 1 so that w can only service one task at a time. Given a set of n tasks $\mathcal{T}$, the OTPD problem can be obtained as a reduction from the TOM problem [22], which has been shown to be NP-hard. The OTPD problem is more complex than the instance considered above. □

In a rigid real-time context, such as a food ordering and delivery service, the system needs to respond the each customer quickly. In addition, the system only maintains information on currently unfinished tasks, which does not enable optimized schedules based on a global scope, i.e., over a long time span. Therefore, when multiple tasks are submitted at the same time, OTPD processes them one by one. On the other hand, OTPD can be easily extended to provide the user with multiple options of schedule for the submitted task such that a higher flexibility is achieved. That is to say, OTPD provides top-k schedules with the highest utilities, and the user can choose the most satisfactory one and have it executed. Due to the space limit, we, however, restrict our discussion only on the single option case.

III. Framework

A. Index Structure

To efficiently solve the OTPD problem, we build indexes on both pickup PoIs and all workers' up-to-date locations.

IR^2-tree index. We build an IR^2-tree [10] on pickup PoIs. Once a new task τ is submitted, we search the IR^2-tree with a location $\tau.l$, a set of keywords $\tau.\phi$, and a radius $\tau.r$. A Boolean Range Query [10] is executed which returns the candidate pickup PoIs w.r.t. τ that satisfy the **pickup selection constraint**.

Grid index. We maintain a grid index to store the up-to-date location of each worker, which is simple and yet efficient for handling updates compared with other indexes (e.g., R-tree and Quadtree). The grid index divides the entire spatial region into equal-sized quad cells and forms a hierarchy of cells. To facilitate identifying candidate workers, we build an inverted list to keep the locations of workers in each leaf cell. When a worker moves, we can easily find the cell the worker is located in, and we can update the corresponding inverted list if necessary without modifying the hierarchical structure of the grid index. Once a new task τ is submitted, we search the grid index with a location $\tau.l$ and a range radius $w.r$, thus finding all candidate workers w.r.t. τ that satisfy the **task acceptance constraint**.

B. System Overview

Fig. 3 shows an overview of the proposed system. The IR^2-tree and the grid index are built on the pickup PoIs and the workers' up-to-date locations, respectively. Each worker's location is continuously sent to the server, and the grid index is updated accordingly in real-time. Once a task is submitted by a user through the communication interface, the system processes it as follows:

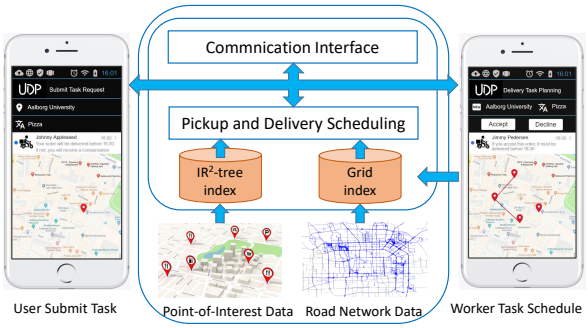


Fig. 3. An Overview of System

1) Candidate workers and pickup PoIs w.r.t. the task are identified by accessing the IR^2-tree and the grid index.
2) The next step involves different scheduling algorithms. For example, SKT (introduced in Section V) accesses the SK-tree that stores selected and skyline schedules of each candidate worker while trying to insert the task into the current schedule.
3) Finally, if a task is successfully inserted into a worker's current schedule, the communication interface sends the schedule information to both the user and the worker. Otherwise, the task is rejected, and the user gets instant feedback. The user can then decide to resubmit the task, possibly with relaxed constraints.

IV. Greedy Assignment Algorithm

We proceed to introduce a greedy assignment algorithm (GA) to handle new tasks online. In GA, we process the upcoming tasks in a greedy manner and assign each new task to the currently best matched worker.

Given a new task τ, we obtain a set of m candidate workers W_τ and a set of n candidate pickup PoIs P_τ by searching two index structures, thus obtaining $m \times n$ worker-item-task matches $\{(w_i, p_j, \tau)\}$, $w_i \in W_\tau, p_j \in P_\tau$. Therefore, we have to enumerate all the permutations. That is, for each w_i, we attempt to insert n pickup delivery event pairs $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ into $\mathcal{DS}(w_i)$. For each event pair $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$, we find the valid insertion positions with the minimum incremental cost. Finally, we assign the task τ to the worker w_i with the maximum utility-cost ratio, which is computed as follows.

$$\Delta \mathsf{ur}_{i,j} = \mu(w_i, p_j, \tau)/\Delta \mathsf{cost}, \tag{8}$$

where Δcost is the incremental cost of successfully inserting $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ into $\mathcal{DS}(w_i)$. If $\hat{\mathcal{DS}}(w_i)$ is the new schedule, we have $\Delta \mathsf{cost} = \mathsf{cost}(\hat{\mathcal{DS}}(w_i)) - \mathsf{cost}(\mathcal{DS}(w_i))$.

A. Inserting a Pickup Delivery Event Pair

To insert an event pair $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ into $\mathcal{DS}(w_i)$, we first need to find valid positions for $x_\tau^{\triangleright}$ and update the information of

all the events after the insertion. Then, we continue to find valid positions for $x_\tau^{\triangleleft}$. Finally, we obtain a new task schedule $\hat{\mathcal{DS}}(w_i)$ with the minimum incremental cost.

1) Valid Event Insertion: In order to insert an event x_τ between two consecutive events x_i and x_{i+1}, we use a function ValidCheck that considers the following conditions:

(i) **Event Reachability Condition.** The event location $x_\tau.l$ must be reachable from $x_i.l$ before deadline $x_\tau.dl$. If we assume no slack time is used for the event x_i, we have: $x_i.t^- + \mathsf{cost}(x_i.l, x_\tau.l) \leq x_\tau.dl$.

(ii) **Maximum Detour Condition.** The cost of the detour from $x_i.l$ through $x_\tau.l$ to $x_{i+1}.l$ should not exceed the latest leaving time of x_{i+1} such that the reachability of all the subsequent events can be secured. Thus, we have: $\mathsf{cost}(x_i.l, x_\tau.l) + \mathsf{cost}(x_\tau.l, x_{i+1}.l) \leq x_{i+1}.t^+ - x_i.t^-$.

(iii) **Capacity Constraint Condition.** Based on the **task capacity constraint**, the task load after a pickup event should not exceed w's capacity, i.e., $|x_i.R| + 1 \leq w.c$.

Instead of traversing all the events to find valid positions, we use the Lemma 1 to prune invalid positions with the event deadline such that the traversal can stop early:

Lemma 1 (**Pruning by Event Deadline**). *To insert a new event x_τ into $\mathcal{DS}(w)$, if the deadline of x_τ is earlier than the earliest arrival time of an event $x_i \in \mathcal{DS}(w)$ i.e., $x_\tau.dl < x_i.t^-$, then the event x_τ cannot be inserted after the event x_i. Since if $x_\tau.dl < x_i.t^-$, w cannot arrive at $x_\tau.l$ before deadline $x_\tau.dl$.*

2) Schedule Update: After inserting an event x_τ into $\mathcal{DS}(w)$, we employ a function UpdateSchedule to update the information of all the events. Assume that x_τ is inserted after x_i, it is obvious that the task loads of all the subsequent events x_i' must be updated. For the earliest arrival time, we observe that only the subsequent events of x_τ are affected; these are thus updated accordingly. For the latest leaving time, in contrast, only the preceding events of x_τ can be affected.

Definition 5 (**Pivot Event**). *For an event x_i, if its latest leaving time equals its deadline, i.e., $x_i.t^+ = x_i.dl$, x_i is a pivot event.*

For two events x_i and x_j, if x_j is the first pivot event after x_i, the latest leaving time of x_i is controlled either by its own deadline $x_i.dl$ or by $x_j.t^+$.

Lemma 2. *Given a new schedule $\mathcal{DS}(w)'$ of inserting x_τ into $\mathcal{DS}(w)$ after x_i, if x_j $(0 \leq j \leq i)$ is a pivot event in both $\mathcal{DS}(w)$ and $\mathcal{DS}(w)'$, the latest leaving time of an event x_k $(k \leq j)$ in $\mathcal{DS}(w)'$ remains unchanged.*

Proof. We know that $x_k.t^+$ is controlled by either $x_k.dl$ or $x_j.t^+$ in $\mathcal{DS}(w)$. As x_j is also a pivot event in $\mathcal{DS}(w)'$, $x_k.t^+$ remains controlled by either $x_k.dl$ or $x_j.t^+$ without a change, which completes the proof. □

Therefore, the update of the latest leaving time of preceding events may stop early due to Lemma 2.

Algorithm 1: Greedy Assignment Algorithm (GA)

Input: τ, $W_\tau = \{w_1, \ldots, w_m\}$, $P_\tau = \{p_1, \ldots, p_n\}$
Output: An updated $\hat{\mathcal{DS}}(w)$ with $LB_{\Delta\mathsf{ur}}$

```
1  LB_Δur ← 0;  DŜ(w) ← ϕ;
2  foreach w_i ∈ W_τ do
3      foreach p_j ∈ P_τ do
4          Compute UB_Δcost and construct (x_τ^▷, x_τ^◁);
5          DŜ(w), UB_Δcost ←
             InsertEventPair(DS(w_i),(x_τ^▷, x_τ^◁), UB_Δcost);
6          if DŜ(w) ≠ ∅ then
7              Update LB_Δur with UB_Δcost and μ(w_i, p_j, τ);
8          end
9      end
10 end
11 return DŜ(w) with LB_Δur;
```

3) Procedure InsertEventPair: Initially, we take an upper bound incremental cost $UB_{\Delta\mathsf{cost}}$ as input to pruning unnecessary insertions, and we maintain a temporary $\mathcal{DS}(w)'$ and $\hat{\mathcal{DS}}(w)$ to store intermediate results. For each inserted position of $x_\tau^{\triangleright}$, procedure InsertEventPair checks if the traversal can stop early based on Lemma 1. If not, we employ the ValidCheck function to determine if inserting $x_\tau^{\triangleright}$ is valid. If so, we check if the incurred cost exceeds the current $UB_{\Delta\mathsf{cost}}$ such that the insertion can be skipped. Otherwise, we insert $x_\tau^{\triangleright}$ and update $\mathcal{DS}(w)'$ with the UpdateSchedule function. Likewise, we check if the traversal can stop early for each inserted position of $x_\tau^{\triangleleft}$, and repeat the same operations for $x_\tau^{\triangleleft}$. If a new schedule with a smaller incremental cost can be found after inserting $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ into $\mathcal{DS}(w)'$, we update $\hat{\mathcal{DS}}(w)$. Finally, the $\hat{\mathcal{DS}}(w)$ with minimum $\Delta\mathsf{cost}$ is returned.

B. Finding Maximum $\Delta\mathsf{ur}$

The GA aims to assign τ to the worker w_i with maximum utility-cost ratio. As previously mentioned, each w_i has n worker-item-task matches $\{(w_i, p_j, \tau)\}$. We denote the lower bound utility-cost ratio by $LB_{\Delta\mathsf{ur}}$. Therefore, for each match (w_i, p_j, τ), we can immediately obtain the upper bound incremental cost $UB_{\Delta\mathsf{cost}}$ using Eq. 8. For a match (w_i, p_j, τ), we attempt to insert the event pair $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ w.r.t. p_j and τ into $\mathcal{DS}(w_i)$. If the incremental cost of this insertion exceeds $UB_{\Delta\mathsf{cost}}$, the match can be discarded.

Algorithm 1 introduces the details of GA. In order to prune the unnecessary workers and matches, we first sort the workers in descending order based on their ratings, and then we sort the pickup PoIs in descending order of their ratings. Therefore, the workers and matches with lower utilities may have a tighter upper bound incremental cost and can be pruned quickly. Lines 2 – 10 insert each event pair into each worker's dynamic schedule one by one using procedure InsertEventPair. Finally, the $\hat{\mathcal{DS}}(w)$ with maximum $\Delta\mathsf{ur}$ is returned.

Complexity Analysis. Given a $\mathcal{DS}(w)$ with an average of $\overline{h}$ events, the cost for permutation is $O(\overline{h}^2)$. Therefore, the overall cost of procedure InsertEventPair is $O(\overline{h}^2)$. Assuming

n_w candidate workers and n_p candidate pickup PoIs, the cost of sorting PoIs is $O(n_p \log n_p)$, and the cost of sorting workers is $O(n_w \log n_w)$. The GA calls procedure InsertEventPair $n_w \times n_p$ times, so the overall cost is $O(n_p \log n_p + n_w \log n_w + n_w \cdot n_p \cdot \overline{h}^2)$. In practice, the GA runs fast due to the pruning power under small n_w, n_p, and $\overline{h}$.

V. SK-Tree based Algorithm

We proceed to introduce a Skyline Kinetic Tree (SK-tree) based algorithm (SKT). For each worker, GA only maintains the currently best schedule and abandons all the other valid schedules, which means that the result quality cannot be guaranteed. For instance, assume that a new task τ is assigned to w with two candidate pickup PoIs p_1 and p_2. As $\Delta\mathsf{ur}_{p_1} > \Delta\mathsf{ur}_{p_2}$, so $\mathcal{DS}(w_i) = (x_0, x_{p_1}, x_\tau)$ is arranged by GA where (x_{p_1}, x_τ) corresponds to p_1 and τ. When another task τ' arrives with two candidate pickup PoIs p_3 and p_4, GA updates $\mathcal{DS}(w_i) = (x_0, x_{p_1}, x_{p_3}, x_\tau, x_{\tau'})$ as $\Delta\mathsf{ur}_{p_3} > \Delta\mathsf{ur}_{p_4}$. However, if p_2 is kept in the schedule in the first round, p_4 would be a better choice than p_3 if p_4 is closer to p_2 with a smaller incremental cost, and $\mu(w_i, p_2, \tau) + \mu(w_i, p_4, \tau')$ is greater than $\mu(w_i, p_1, \tau) + \mu(w_i, p_3, \tau')$.

Therefore, we propose to keep intermediate computation results to enable a tradeoff between space and result quality. It is impractical to store all valid schedules due to the exponential space consumption. Assume that w has a set of unfinished tasks T, the same T may correspond to different pickup PoI sets P. For each assignment $\mathcal{A}(w, P, T)$, we only store the schedule with the current minimum cost, as all valid schedules of $\mathcal{A}(w, P, T)$ have the same utility. For all the schedules of different assignments, we select the one with the maximum overall utility to be executed by the server, denoted as $\mathcal{DS}(w)_0$, and we keep its skyline schedules on the dimensions of utility and cost in the SK-tree. Therefore, an SK-tree is built for each worker w on $\mathcal{DS}(w)_0$ and its skylines.

Given a new task, we perform a two-level best-first-search (L2BFS) on the SK-tree of each candidate worker to find the valid schedules. It is worth noting that only the SK-tree of best matched worker stores the newly selected schedule and its skylines from the intermediate computation results. We do not consider a globally optimal scope due to the a practical concern that once a task is assigned, both the customer and the worker would not like to accept a deletion or switch operation on a scheduled task.

A. Skyline Kinetic Tree

Specifically, an SK-tree captures a selected schedule and its skylines, i.e., $\mathcal{SK}(w) = \{\mathcal{DS}(w)_0, \mathcal{DS}(w)_1, \ldots, \mathcal{DS}(w)_k\}$. When the worker moves on $\mathcal{DS}(w)_0$, some skyline schedules that do not share common events with the selected schedule may become obsolete. The root of the SK-tree always tracks the up-to-date location of the worker, and it becomes an acceptance event x_0 when a new task is assigned. In a dynamic schedule $\mathcal{DS}(w)$, an array is maintained to keep an average number of $\overline{h}$ events. In an SK-tree, each root-to-leaf path represents a valid dynamic schedule of w, and each node x_i stores the information of an event. In addition, each node x_i keeps the information $x_i.sid$, which is the ID of the schedule $\mathcal{DS}(w)_{x_i.sid}$ that has the maximum utility among all schedules covering x_i. The space cost of $\mathcal{SK}(w)$ is $O(k \cdot \overline{h})$.

B. Inserting a New Task

Once a new task τ arrives, our approach traverses the SK-tree of each candidate worker by performing an L2BFS with different event pairs. Assume τ corresponds to a set of n candidate pickup PoIs P_τ for each candidate worker w_i. Therefore, for each w_i, we attempt to insert n event pairs into w_i's SK-tree. For each $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$, we aim to find an assignment with maximum utility and a corresponding schedule with minimum cost. Then we compare these new schedules of all n event pairs and select one and its skylines. Finally, we assign τ to the worker whose selected schedule has maximum utility.

To employ an L2BFS on the SK-tree of a candidate worker w, we utilize two priority queues $PQ^{\triangleright}$ and $PQ^{\triangleleft}$ to hold valid inserted positions for $x_\tau^{\triangleright}$ and $x_\tau^{\triangleleft}$, respectively. To prune the unnecessary positions, we initialize a lower bound utility LB_μ and an upper bound cost UB_{cost} of a potential valid schedule. As the incremental utility is the same for all schedules, LB_μ is used to keep the original schedule utility.

Upper bound utility at level one. To insert $x_\tau^{\triangleright}$ at an edge $e(i, j)$ between the adjacent nodes x_i and x_j, we insert $e(i, j)$ into $PQ^{\triangleright}$ based on the utility of the schedule $\mathcal{DS}(w)_{x_j.sid}$, denoted as $\mathcal{U}(e(i, j))$. If node x_i is a leaf node, we insert a dummy edge $e(i, \infty)$ based on the utility of $\mathcal{DS}(w)_{x_i.sid}$.

Lemma 3. *The upper bound utility of a potential schedule with $x_\tau^{\triangleright}$ being inserted at $e(i, j)$ is $\mathcal{U}(e(i, j))$.*

Proof. As $x_\tau^{\triangleright}$ is inserted at $e(i, j)$, $\mathcal{DS}(w)_{x_j.sid}$ is the schedule with the maximum utility that covers $e(i, j)$. If $x_\tau^{\triangleleft}$ is inserted at an edge that is also covered by $\mathcal{DS}(w)_{x_j.sid}$, the utility is $\mathcal{U}(e(i, j))$. Otherwise, if $x_\tau^{\triangleleft}$ is inserted in other branches, the utility is smaller than $\mathcal{U}(e(i, j))$. □

Let $e(i, j)$ be the top of $PQ^{\triangleright}$. From Lemma 3, if $\mathcal{U}(e(i, j))$ is smaller than LB_μ, we are unable to find a schedule with a higher utility, so the traversal on $\mathcal{SK}(w)$ stops. Otherwise, we insert $x_\tau^{\triangleright}$ at $e(i, j)$ and attempt to insert $x_\tau^{\triangleleft}$ at the new subtree rooted at $x_\tau^{\triangleright}$.

Upper bound utility at level two. We insert the candidate edges into $PQ^{\triangleleft}$ based on the utilities. If we insert $x_\tau^{\triangleright}$ and $x_\tau^{\triangleleft}$ into w's SK-tree at $e(i, j)$ and $e(i', j')$ respectively, the maximum utility of the schedule that covers $e(i, j)$ and $e(i', j')$ is $\mathcal{U}(e(i', j'))$. This holds because the utility of the schedule where $(x_\tau^{\triangleright}, x_\tau^{\triangleleft})$ are inserted is determined by $e(i', j')$, and $\mathcal{U}(e(i', j'))$ is the maximum utility of the schedules covering $e(i', j')$.

Therefore, once we insert $x_\tau^{\triangleright}$ and $x_\tau^{\triangleleft}$ at $e(i, j)$ and $e(i', j')$, respectively, to compare with the current LB_μ, we have:

Lemma 4. *A candidate schedule is found iff $LB_\mu \leq \mathcal{U}(e(i', j'))$, and we can update LB_μ and UB_{cost} accordingly.*

Proof. To compare LB_μ and $\mathcal{U}(e(i', j'))$, we have:

- If $LB_\mu < \mathcal{U}(e(i',j'))$, we update LB_μ, which means that we have found a schedule with higher utility. Then we set UB_{cost} to $\mathsf{cost}(\mathcal{DS}(w)_{x'_j.sid}) + \Delta\mathsf{cost}$.
- If $LB_\mu = \mathcal{U}(e(i',j'))$, we know that we are examining the same schedule, and we just need to check whether we can find a position with smaller cost. If $\mathsf{cost}(\mathcal{DS}(w)_{x'_j.sid}) + \Delta\mathsf{cost} \leq UB_{\mathsf{cost}}$, we update UB_{cost}; otherwise, we continue to examine the next edge for $x_\tau^\triangleleft$ in $PQ^\triangleleft$.
- If $LB_\mu > \mathcal{U}(e(i',j'))$, we know that we cannot find a better position for $x_\tau^\triangleleft$ when inserting $x_\tau^\triangleright$ at $e(i,j)$, so we continue to examine the next edge for $x_\tau^\triangleright$ in $PQ^\triangleright$.

□

C. Algorithm SKT

Algorithm 2 shows the details of SKT. To initialize, we maintain a competitor set C to hold skyline schedules for a candidate worker. Then we enumerate the workers and PoIs as discussed in the context of GA. Line 3 initializes a set C_{tmp} to temporarily keep skylines for each w_i.

1) Pruning at Level One: For $x_\tau^\triangleright$, we insert the edges starting at the root of $\mathcal{SK}(w_i)$ into $PQ^\triangleright$. While $PQ^\triangleright$ is not empty, we pop the top edge $e(i,j)$ and insert its sub-edges into $PQ^\triangleright$. We first check the validity of inserting $x_\tau^\triangleright$ at $e(i,j)$. Then we compare $\mathcal{U}(e(i,j))$ with the current LB_μ to see if we are able to find a schedule with a higher utility.

2) Pruning at Level Two: After we insert $x_\tau^\triangleright$ at $e(i,j)$, we continue to extract the subtree rooted at $x_\tau^\triangleright$ from $\mathcal{SK}(w_i)$ and update the information. Likewise, $PQ^\triangleleft$ is maintained to keep the candidate edges for $x_\tau^\triangleleft$ during the traversal of the subtree. We initialize $PQ^\triangleleft$ by inserting the edges starting at the root of the new subtree. While $PQ^\triangleleft$ is not empty, we employ a similar way of checking the validity for inserting $x_\tau^\triangleleft$ at $e(i',j')$. Then, we obtain the maximum utility of the schedule covering $e(i,j)$ and $e(i',j')$. Finally, we compare it with LB_μ based on Lemma 4.

If we obtain a new schedule with the maximum utility and minimum cost after traversing $\mathcal{SK}(w_i)$ w.r.t. p_j, we compare it with the existing new schedules w.r.t. $P_\tau - \{p_j\}$ in C_{tmp} and update the selected schedule $\mathcal{DS}(w_i)_0$ and its skylines. Finally, we assign τ to the best matched worker whose $\mathcal{DS}(w)_0$ has the highest utility.

Complexity Analysis. Assume that $\mathcal{SK}(w)$ is built by $\overline{k}$ $\mathcal{DS}(w)$ with $\overline{h}$ levels. In the worst case, the cost is $O(\overline{k} \cdot \overline{h}^2)$ to insert an event pair. Therefore, the SKT has time complexity $O(n_p \log n_p + n_w \log n_w + n_w \cdot n_p \cdot \overline{k} \cdot \overline{h}^2)$.

Algorithm 2: SK-tree Based Algorithm (SKT)

Input: τ, $W_\tau = \{w_1, \ldots, w_m\}$, $P_\tau = \{p_1, \ldots, p_n\}$
Output: An updated $\hat{\mathcal{SK}}(w)$

```
1  Initialize a competitor set C;
2  foreach w_i ∈ W_τ do
3      Initialize a competitor set C_tmp;
4      foreach p_j ∈ P_τ do
5          Initialize and PQ^▷;
6          while PQ^▷ is not empty do
7              Find a valid e(i,j) from PQ^▷;
8              initialize PQ^◁ with root edges of subtree of x_τ^▷;
9              while PQ^◁ is not empty do
10                 Find a valid e(i',j') from PQ^◁;
11             end
12         end
13         Compare and insert the new schedule into C_tmp;
14     end
15     Update C with C_tmp if necessary;
16 end
17 Merge the schedules in C and form SK̂(w);
18 return SK̂(w);
```

VI. Processing Tasks in Batch

The main problem of GA and SKT is that they have to enumerate all candidate workers and pickup PoIs, which has high computational costs even if they can achieve high overall utility. Therefore, we proceed to develop a density-based grouping algorithm (DG) that processes the tasks from a short timespan in batch. Specifically, we partition the tasks into groups based on the density of the task destinations in a region, and we select a pickup PoI for each task in a group such that the maximum distance between any two pickup PoIs is minimized. Intuitively, two tasks in a same group are likely to have both close pickup and delivery locations such that it is more likely that they can be assigned to the same worker with low incremental cost. In addition, the pre-selection of pickup PoIs avoids enumeration all potential PoIs for each candidate worker, which reduces the computational costs.

A. Grouping Tasks

To divide the tasks into groups, we first partition the space based on the tasks' destinations using a quadtree. Once the number of tasks in a bucket reaches a capacity limit, a task group is obtained. For each group, as a task may correspond to multiple pickup PoIs, we select a pickup PoI for each task with the goal of minimizing the maximum distance between any two pickup PoIs in a group. Thus if two tasks of a group, i.e., τ_1, τ_2, are assigned to the same worker w, the transfer distance that w must travel between τ_1 and τ_2 is bounded by the maximum distance such that more tasks can be served and the overall utility can be improved.

1) Partitioning Tasks by Quadtree: Given a batch T with n tasks, we partition the tasks into groups $\{G_1, \ldots, G_k\}$ by constructing a quadtree. Afterwards, we take the tasks in each leaf node as a group G_i. Specifically, we initialize a bucket that covers the whole space. Then, we sequentially insert the destination of each task into the bucket. The maximum capacity of each bucket is set to a constant N, which is the group density. Once N is reached in a bucket, it splits into four sub-buckets. The procedure continues until all the tasks have been processed. The tree directory follows the hierarchical decomposition of the quadree. If the tasks' destinations are distributed evenly, each insertion takes $O(\log n)$ time. Therefore, the task grouping takes $O(n \log n)$ time.

2) Selecting Pickup PoIs: In a group G_i, as $\tau_j \in G_i$ is possibly associated with multiple candidate pickup PoIs, we

Algorithm 3: Procedure SelectPickup()

Input: G, the candidate set P_{τ_i} for each $\tau_i \in G$
Output: $\hat{P}$ with minimum $\varphi(\hat{P})$

1 $\tau_m \leftarrow \tau_i \in G$ with least candidate pickup PoIs;
2 initialize a map $minD$ with $G-1$ key-value pairs, $\hat{P} \leftarrow \phi$;
3 **foreach** $\tau' \in G - \tau_m$ **do**
4 $\quad minD[\tau'] = +\infty$
5 **end**
6 **foreach** $p_j^{\tau_m} \in P_{\tau_m}$ **do**
7 $\quad P \leftarrow p_j^{\tau_m}$;
8 $\quad$ **foreach** $p_k^{\tau'} \in \cup_{\tau' \in G-\tau_m} P_{\tau'}$ **do**
9 $\quad\quad$ compute $\mathsf{dist}(p_j^{\tau_m}.l, p_k^{\tau'}.l)$;
10 $\quad\quad$ **if** $\mathsf{dist}(p_j^{\tau_m}.l, p_k^{\tau'}.l) < minD[\tau']$ **then**
11 $\quad\quad\quad$ $minD[\tau'] = \mathsf{dist}(p_j^{\tau_m}.l, p_k^{\tau'}.l)$
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 $\quad P \leftarrow$ all PoIs have the values in $minD$;
15 $\quad$ compute $\varphi(P)$;
16 $\quad$ **if** *$\hat{P}$ is empty or* $\varphi(P) < \varphi(\hat{P})$ **then**
17 $\quad\quad \hat{P} \leftarrow P$;
18 $\quad$ **end**
19 **end**
20 **return** $\hat{P}$;

employ a pre-selection to assign a single PoI to each τ_j. By doing so, we skip the PoI enumeration in the scheduling process. Assume that in group G_i, we select a set of pickup PoIs P, where each τ_j has a corresponding pickup PoI $p_j \in P$, the pickup diameter of P is defined as the maximum distance between any two pickup PoIs, i.e., $\varphi(P) = \max_{p_1,p_2 \in P} \mathsf{dist}(p_1.l, p_2.l)$. The pickup PoI selection aims to find a P such that $\varphi(P)$ is minimized.

Theorem 2. *Finding the P with minimum $\varphi(P)$ is NP-hard.*

Proof. We prove the theorem by a reduction from the 3-SAT problem. An instance of the 3-SAT problem consists of $\phi = C_1 \wedge \cdots \wedge C_n$, where each clause $C_i = \{x_i \vee y_i \vee z_i\}(i = 1, \ldots, n)$, and $\{x_i, y_i, z_i\} \subset \{u_1, \overline{u}_1, \ldots, u_m, \overline{u}_m\}$. The decision problem is to determine whether we can assign a Boolean value to each variable u_i such that ϕ is true. We transform an instance of the 3-SAT problem to an instance of finding an optimal P as follows. We consider a circle of diameter d'. Two PoIs u_i and $\overline{u}_i$ are placed diametrically on the circle, and the distance between u_i and $\overline{u}_i$ is d'. We set $d' = d + \epsilon$, where ϵ is a sufficiently small and positive value, such that the distance between any two pickup PoIs corresponding to different variables is at most d. We create a task $\tau_i (i \in [1, m])$ and associate it with the PoIs w.r.t. u_i and $\overline{u}_i$. For each C_i, we create a task $\tau_{m+i} (i \in [1, n])$ and associate it with the pickup PoIs of the three variables in C_i. If the instance of finding an optimal P can be solved then 3-SAT instance can also be solved. This completes the proof. □

B. Algorithm DG

Theorem 2 implies that it is impractical to find an optimal P. Therefore, we instead propose a procedure SelectPickup() to select the pickup PoIs in a greedy manner, as shown in Algorithm 3. The basic idea is as follows: we first find a task $\tau_m \in G$ with the least number of candidate PoIs. Then, around each pickup PoI $p_j^{\tau_m}$ of τ_m, for each task $\tau' \in G - \tau_m$, we find the closest pickup PoI of each τ' in Lines 3–12. After all pickup PoIs of τ_m are processed, we select the set P that has the smallest $\varphi(P)$ for the tasks in G in Lines 13–14. We know that each group has fewer than N tasks, and we assume that the average number of pickup PoIs for each task is n_p. For each $p_j^{\tau_m}$, it then takes $O(N \cdot n_p)$ time to find a candidate P. Therefore, the procedure SelectPickup() takes $O(n_p^2)$ time, as N is a constant. Having partitioned the tasks into groups, we process the groups in descending order of their overall utilities. For each group, we apply the same task inserting method as in GA and SKT to assign the tasks.

Complexity Analysis. The task grouping costs $O(n \log n)$, and procedure SelectPickup() costs $O(n_p^2)$ for each G_i, and we employ an insertion for each task. In DG, each task has only one candidate pickup PoI; thus, the task insertion cost is $\mathsf{f} = O(n_w \log n_w + n_w \cdot \overline{h}^2)$. Therefore, DG takes $O(n \log n + n \cdot n_p^2 + n \cdot \mathsf{f})$ time.

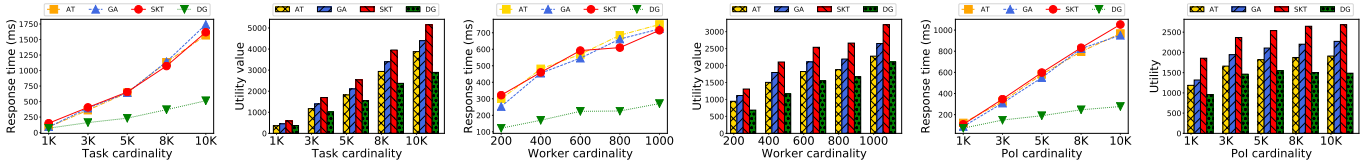
VII. Experiments

A. Experimental Settings

All algorithms were implemented in Java on Linux and run on an Intel(R) CPU i7-4770@3.4GHz and 32G RAM.

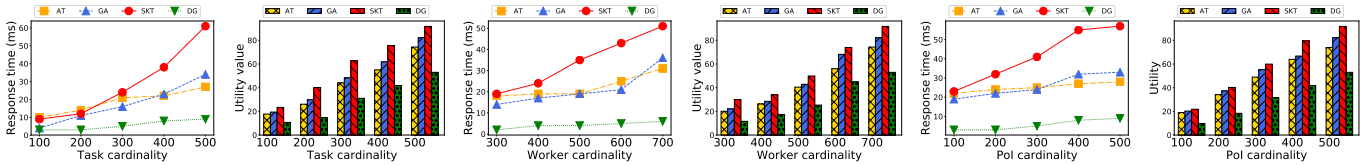
Datasets and Parameter settings. We use two datasets, a real dataset called gMission [22] that is generously provided to us, and a synthetic dataset. Dataset gMission contains 532 tasks, 713 workers, and 532 pickup PoIs. Each task has a description, a destination, a submission time, a set of keywords describing the desired items, an expiration time, and an extra reward. Each worker also has a location, a release time, and a rating. Each pickup PoI has a location, a set of keywords, and a rating. As the original dataset does not contain PoI information, we generate a set of PoIs whose cardinality is the same as that of the task set such that each task corresponds to one PoI on average. All tasks share a unified task radius, and all workers have a unified radius and capacity. The synthetic dataset is generated by using an existing tool [18]. We generate the scores and locations to have Normal and Exponential distribution respectively. We set the default task cardinality to 5K to investigate the scalability of the proposed algorithms. The parameter settings are shown in Table II.

Algorithms. We evaluate the performance of four algorithms: the adaptive threshold algorithm (AT) [22], the greedy assignment algorithm (GA), the skyline kinetic tree-based algorithm (SKT), and the density-based grouping algorithm (DG) according to two metrics: response time (ms) and utility. Specifically, to fairly compare with the AT algorithm [22], we first adapt AT to assign the task and then call the procedure InsertEventPair to find the schedule with smallest incremental cost.



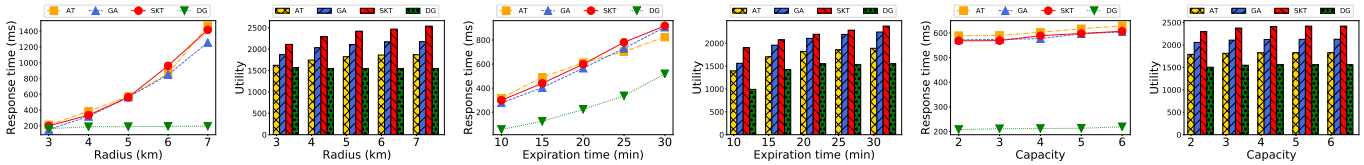
(a) Effect of $|T|$ (b) Effect of $|T|$ (c) Effect of $|W|$ (d) Effect of $|W|$ (e) Effect of $|P|$ (f) Effect of $|P|$

Fig. 4. Effect of task, worker, pickup PoI cardinality on synthetic data



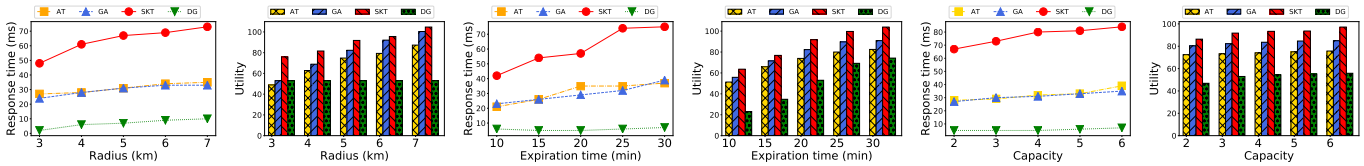
(a) Effect of $|T|$ (b) Effect of $|T|$ (c) Effect of $|W|$ (d) Effect of $|W|$ (e) Effect of $|P|$ (f) Effect of $|P|$

Fig. 5. Effect of task, worker, pickup PoI cardinality on real data



(a) Time on syn data (b) Utility on syn data (c) Time on syn data (d) Utility on syn data (e) Time on syn data (f) Utility on syn data

Fig. 6. Effect of radius, expiration time, and capacity on response time (ms) and utility on synthetic data



(a) Time on real data (b) Utility on real data (c) Time on real data (d) Utility on real data (e) Time on real data (f) Utility on real data

Fig. 7. Effect of radius, expiration time, and capacity on response time (ms) and utility on real data

TABLE II
PARAMETER SETTINGS

Parameters	Values
Task cardinality (synthetic dataset)	1K, 3K, **5K**, 8K, 10K
Worker cardinality (synthetic dataset)	200, 400, **600**, 800, 1000
Pickup PoI cardinality (synthetic dataset)	1K, 3K, **5K**, 8K, 10K
Task cardinality (real dataset)	100, 200, 300, 400, **500**
Worker cardinality (real dataset)	300, 400, 500, 600, **700**
Pickup PoI cardinality (real dataset)	100, 200, 300, 400, **500**
Task worker radius (km)	3, 4, **5**, 6, 7
Expiration (waiting) time (min)	10, 15, **20**, 25, 30
Capacity	2, **3**, 4, 5, 6
α, β	(0.1,0.1),(0.2,0.2),**(0.3,0.3)**,(0.4,0.4)
Group density N	10, 20, **30**, 40, 50
Batch length	10s, **30s**, 60s, 90s, 120s

B. Efficiency and Effectiveness Evaluation

Effect of task cardinality. We study the effect of task cardinality on the performance. In the synthetic dataset, we enlarge $|T|$ from 1K to 10K and use 5K as the default task cardinality to show the scalability of proposed algorithms. Likewise, in the real dataset, we enlarge $|T|$ from 100 to 500 and use 500 as the default task cardinality. These sub-datasets are all randomly sampled. As shown in Figs. 4(a), 4(b), 5(a), and 5(b), both the response time and utility increase when we enlarge the task cardinality. It is easy to understand that when more tasks are submitted, the proposed algorithms can achieve higher utility and take more time to perform the scheduling. For the response time, DG is the most efficient one among them. For the utility, SKT is most effective with the highest utility.

Effect of worker cardinality. In the synthetic dataset, we enlarge the worker cardinality $|W|$ from 200 to 1000, and use 600 as the default value in the remaining experiments. Likewise, in the real dataset, we enlarge $|W|$ from 300 to 700, and use 700 as the default value. As shown in Figs. 4(c), 5(c), 4(d), and 5(d), the response time and utility exhibit increasing trends similarly to what was observed in the task cardinality experiment. This is because more workers are available to accept tasks. In terms of response time, DG outperforms GA,

AT, and SKT. Next, SKT has a higher utility than GA and AT, and all outperform DG.

Effect of pickup PoI cardinality. In the synthetic dataset, we enlarge $|P|$ from 1K to 10K and use 5K as the default value in the remaining experiments. In the real dataset, we enlarge $|P|$ from 100 to 500 and use 500 as the default value. As shown in Figs. 4(e), 4(f), 5(e), and 5(f), SKT achieves the highest utilities, which is not surprising. DG is less effective than SKT, AT, and GA, and it takes much less time than SKT, AT, and GA.

Effect of radius. To study the effect of the task and worker radii on the performance, we vary it from 3 km to 7 km. As shown in Figs. 6(a), 6(b), 7(a), and 7(b), the response times of the algorithms increase with larger radius, and the overall utilities of the algorithms on both datasets also increase. The reason is that when we have larger radii, more workers will locate in the feasible areas of the tasks, and the tasks also have more pickup PoIs for items. Therefore, the algorithms can assign the tasks to workers and pickup PoIs with higher ratings. SKT achieves the highest utilities on both the synthetic and real datasets. In terms of response time, DG is better than GA, AT, and SKT. When the radii are set to a small value, the difference in response time between the algorithms is not obvious. However, it becomes substantial when we use large radii. We choose 5 km as the default setting.

Effect of expiration time. To study the effect of expiration (waiting) time on the performance, we vary the duration from 10 mins to 30 mins. As shown in Figs. 6(c), 6(d), 7(c), and 7(d), the overall utilities of the algorithms on both datasets increase when we enlarge the waiting time. This is because when the waiting time gets longer, the workers have more options of which tasks to select. Therefore, our algorithms can assign the tasks to workers with higher overall utilities. Not surprisingly, SKT achieves the highest utilities on both the synthetic and the real datasets. AT even achieves lower utilities than GA. The response time increases as we enlarge the time duration. DG is the fastest and takes less than half of the time that AT, GA, and SKT take. We use 20 minutes as the default setting in all other experiments.

Effect of capacity. To study the effect of the worker capacity on the performance, we vary $w.c$ from 2 to 6. As shown in Figs. 6(e), 6(f), 7(e), and 7(f), the utilities of the algorithms increase slightly on both the synthetic and the real datasets, and the response times of the algorithms also increase. The reason is that a higher capacity allows the workers to accept more tasks, thus increasing the number of tasks assigned and the computation costs. SKT achieves the highest utilities on both the synthetic and the real datasets. GA performs better than AT: their time costs are almost the same, but the utility of GA exceeds that of AT. We choose 3 as the default setting.

Effect of utility parameters. We proceed to study the effect of utility parameters α, β, and $1-\alpha-\beta$ on the performance. Due to the space limitation, we consider only 4 sets of values: (0.1,0.1,0.8), (0.2,0.2,0.6), (0.3,0.3,0.4), and (0.4,0.4,0.2). As can be seen in Fig. 8, in both the synthetic and the real datasets, the response times of our proposed algorithms do not change substantially across the different settings. When we enlarge the values of α, β, the utility values increase in both the synthetic and the real datasets. This is because the rewards of submitted tasks are the same for all the algorithms, while α and β are related to the workers and pickup PoIs that are selected by our algorithms. Therefore, for the default setting, we choose (0.3,0.3,0.4) as the utility parameters. Considering response time, SKT is better than AT and GA on the synthetic dataset, but is slightly worse on the real dataset. DG unsurprisingly is overall fastest.

C. Performance Study on DG Algorithm

We evaluate the performance of DG on synthetic dataset by varying two parameters: group density N, and batch length.

Effect of N. As shown in Fig. 9(a), we vary N from 10 to 50. When N is 30, the lowest cost is achieved. A smaller N costs more due to task partitioning, and a larger N costs more due to pickup PoI selection. The utility values for all the settings of N are close, so we choose 30 as the default setting for DG.

Effect of batch length. As shown in Fig. 9(b), we vary the batch length from 10s to 120s. The results show that a batch length of 30s has the least response time and that the utility is insensitive to the batch length. In addition, batches that are too long are less acceptable to customers that do not want to wait for long to get their tasks assigned. Therefore, we use 30s as the default batch length.

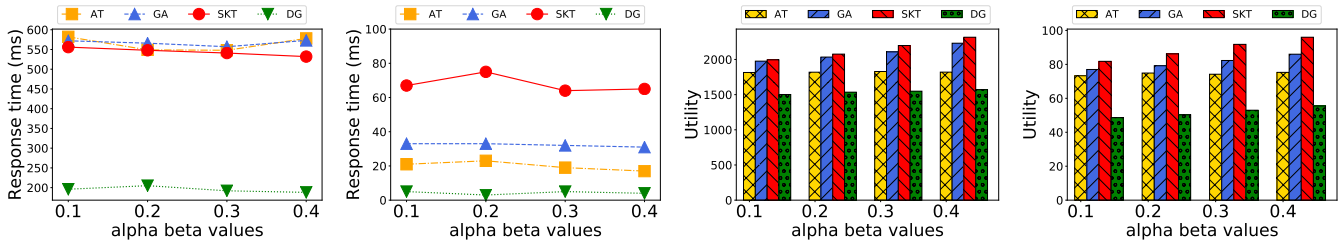
VIII. Related Work

A. Ridesharing

As an instance of the Pickup and Delivery Problem, the ridesharing has gained increasing attention in recent years. Multiple ridesharing systems are developed [15], [21]. In one study [16], in order to provide real-time responses, trip requests with waiting and service time constraints are served in a sequential style, such that for each driver, the total travel time of the ride-sharing schedule is bounded by a set range versus the travel time without sharing. Another study [12] investigates ridesharing with social affinity concerns. The goal is to let riders with high social affinity take the same vehicle, to improve their experience. We study the similar problem of maximizing the utility of task assignment. However, our problem differs as it considers a trichromatic matching utility.

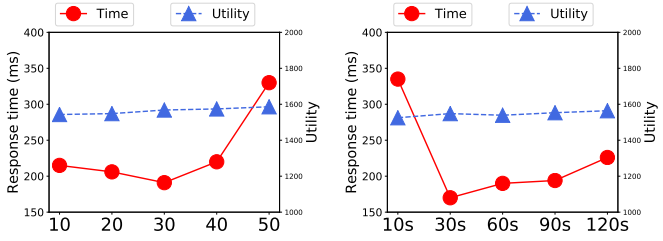
B. Spatial Crowdsourcing

The Pickup and Delivery Problem can be considered as a spatial crowdsourcing application [9], [19], [24], where spatial task assignment is a closely related topic. For static scenarios, the assignment problem can be regarded as the classical maximum weighted bipartite graph matching problem [14], [18]. However, this is not suitable for dynamic scenarios where tasks and workers appear dynamically and the full bigraph is not known in advance. One study [25] considers the online minimum bipartite matching problem in real time spatial data (OMBM), where workers are predefined and tasks arrive on a real-time basis. Finally, another study [22] proposes methods for online trichromatic matching with quality guarantees. A



(a) Time on syn data (b) Time on real data (c) Utility on syn data (d) Utility on real data

Fig. 8. Effect of α, β, and $1-\alpha-\beta$ on response time (ms) and utility



(a) Effect of N (b) Effect of batch length

Fig. 9. Effect of N and batch length of DG on response time (ms) and utility on synthetic dataset

trichromatic matching between worker, workplace, and task is developed. This relates to our matching between worker, pickup PoI, and task. However, our work differs from this approach because each task is associated with multiple pickup PoIs and we consider not only the matching, but also the scheduling for each worker.

IX. Conclusion and Future Work

We study the OTPD problem, where, given a set of crowd workers, pickup PoIs, and pickup and delivery tasks, the goal is to find a delivery schedule formed by trichromatic matches for each worker such that the overall utility is maximized, subject to spatial, temporal, keyword, and capacity constraints. Several algorithms are proposed to solve the OTPD problem efficiently. The empirical studies show that SKT outperforms both GA and AT in terms of utility, but only takes slightly more time than GA and AT. DG processes tasks in batches such that the efficiency is improved substantially when compared to GA, AT, and SKT. Several directions for future research are promising. First, the customer may prefer an interactive query model for OTPD, which may enable understanding the customer's preferences better. Second, it is of interest to conduct a user study involving real customers to demonstrate the satisfaction of the proposed approach.

Acknowledgment

This research is partially supported by NSFC (Grants No. 61902134, 61972069, 61836007, 61832017, 61532018, 61572215), and the Fundamental Research Funds for the Central Universities (HUST: Grants No. 2019kfyXKJC021, 2019kfyXJJS091).

References

[1] Didi: https://www.xiaojukeji.com, 2019.
[2] Ele: https://www.ele.me, 2019.
[3] Gigwalk: https://www.gigwalk.com, 2019.
[4] Meituan: http://waimai.meituan.com, 2019.
[5] Uber, 2019.
[6] Ubereats: https://www.ubereats.com, 2019.
[7] Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. Dynamic pickup and delivery problems. *EJOR*, 202(1):8–15, 2010.
[8] Samantha Bomkamp and Chicago Tribune. Restaurant food delivery heating up. *Columbian.com*, 2016.
[9] Lei Chen and Cyrus Shahabi. Spatial crowdsourcing: Challenges and opportunities. *IEEE Data Eng. Bull.*, 39(4):14–25, 2016.
[10] Lisi Chen, Gao Cong, Christian S Jensen, and Dingming Wu. Spatial keyword query processing: an experimental evaluation. *PVLDB*, 6(3):217–228, 2013.
[11] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S. Jensen. Price-and-time-aware dynamic ridesharing. In *ICDE*, pages 1061–1072, 2018.
[12] Peng Cheng, Hao Xin, and Lei Chen. Utility-aware ridesharing on road networks. In *SIGMOD*, pages 1197–1210, 2017.
[13] Brian Coltin. *Multi-Agent Pickup And Delivery Planning With Transfers*. PhD thesis, 2014.
[14] Ju Fan, Guoliang Li, Beng Chin Ooi, Kian-Lee Tan, and Jianhua Feng. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*, pages 1015–1030, 2015.
[15] Gyozo Gidofalvi, Torben Bach Pedersen, Tore Risch, and Erik Zeitler. Highly scalable trip grouping for large-scale collective transportation systems. In *EDBT*, pages 678–689, 2008.
[16] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 7(14):2017–2028, 2014.
[17] Ece Kamar and Eric Horvitz. Collaboration and shared plans in the open world: Studies of ridesharing. In *IJCAI*, volume 9, page 187, 2009.
[18] Leyla Kazemi and Cyrus Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *ACM SIGSPATIAL GIS*, pages 189–198, 2012.
[19] Guoliang Li, Jiannan Wang, Yudian Zheng, and Michael J. Franklin. Crowdsourced data management: A survey. *TKDE*, 28(9):2296–2319, 2016.
[20] Yafei Li, Rui Chen, Lei Chen, and Jianliang Xu. Towards social-aware ridesharing group query services. *IEEE TSC*, 10(4):646–659, 2017.
[21] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*, pages 410–421, 2013.
[22] Tianshu Song, Yongxin Tong, Libin Wang, Jieying She, Bin Yao, Lei Chen, and Ke Xu. Trichromatic online matching in real-time spatial crowdsourcing. In *ICDE*, pages 1009–1020, 2017.
[23] Na Ta, Guoliang Li, Tianyu Zhao, Jianhua Feng, Hanchao Ma, and Zhiguo Gong. An efficient ride-sharing framework for maximizing shared route. *TKDE*, 30(2):219–233, 2018.
[24] Yongxin Tong, Lei Chen, and Cyrus Shahabi. Spatial crowdsourcing: Challenges, techniques, and applications. *PVLDB*, 10(12):1988–1991, 2017.
[25] Yongxin Tong, Jieying She, Bolin Ding, Lei Chen, Tianyu Wo, and Ke Xu. Online minimum matching in real-time spatial data: experiments and analysis. *PVLDB*, 9(12):1053–1064, 2016.